Orient Academies

A Machine Learning Approach to Multi-Join Query Optimization in Large-Scale Relational Databases

Jalel Messaoudi[1], Nacim Touahria[2]

Abstract

Business-to-consumer digital retail platforms generate extensive observational traces of browsing, search, transaction, messaging, and support interactions, which can be integrated into customer 360 representations. These representations combine identifiers, event histories, inferred preferences, and contextual attributes into longitudinal profiles capable of supporting targeted personalization. Despite the availability of such data, many operational strategies remain based on response prediction or heuristic segmentation, which can systematically conflate correlation with causal impact and lead to inefficient use of incentives, exposure, and capacity. This paper examines a technical framework for causal inference and uplift modeling built directly on integrated customer 360 data with the objective of estimating heterogeneous treatment effects and deploying stable, auditable targeting policies. The discussion focuses on definition of exposure units, temporal alignment of features and outcomes, assumptions for identification in mixed experimental and observational regimes, and the use of orthogonal, doubly robust, and policy-learning methods that operate under budget and operational constraints. Attention is given to the interaction between model structure, identity resolution strategies, and multi-channel treatment assignment, as well as to mechanisms for drift detection, overlap monitoring, and fairness-aware analysis. The framework is intended to be implementable in production environments that require strict latency, governance, and privacy controls, while remaining explicit about assumptions and sensitivities. The paper is descriptive rather than promotional, outlining a set of consistent design choices and analytical components that can be combined to support cautious deployment of causal personalization in B2C digital retail platforms.

[1]Université de Jendouba, Department of Computer Science, Avenue Farhat Hached 41, Jendouba 8100, Tunisia
[2]Université de Naama, Department of Computer Science, Route El-Meridj 22, Naama 45000, Algeria

Contents

1. Introduction

Relational database management systems remain a central platform for transactional and analytical workloads, and their performance depends critically on the quality of the query optimizer [1]. Cost-based optimizers search over alternative execution plans, estimate the runtime cost of each candidate, and select a plan with low predicted cost. For single-table queries and simple joins, the search space is limited and classical techniques can identify reasonable plans with modest effort. However, complex analytical queries often involve many joins, nested subqueries, and correlated predicates. In such cases, the space of admissible join orders grows combinatorially with the number of relations, and exact search quickly becomes infeasible. Optimizers therefore combine dynamic programming for smaller join graphs with heuristics and pruning strategies for larger ones, which can cause the selected plan to deviate noticeably from the

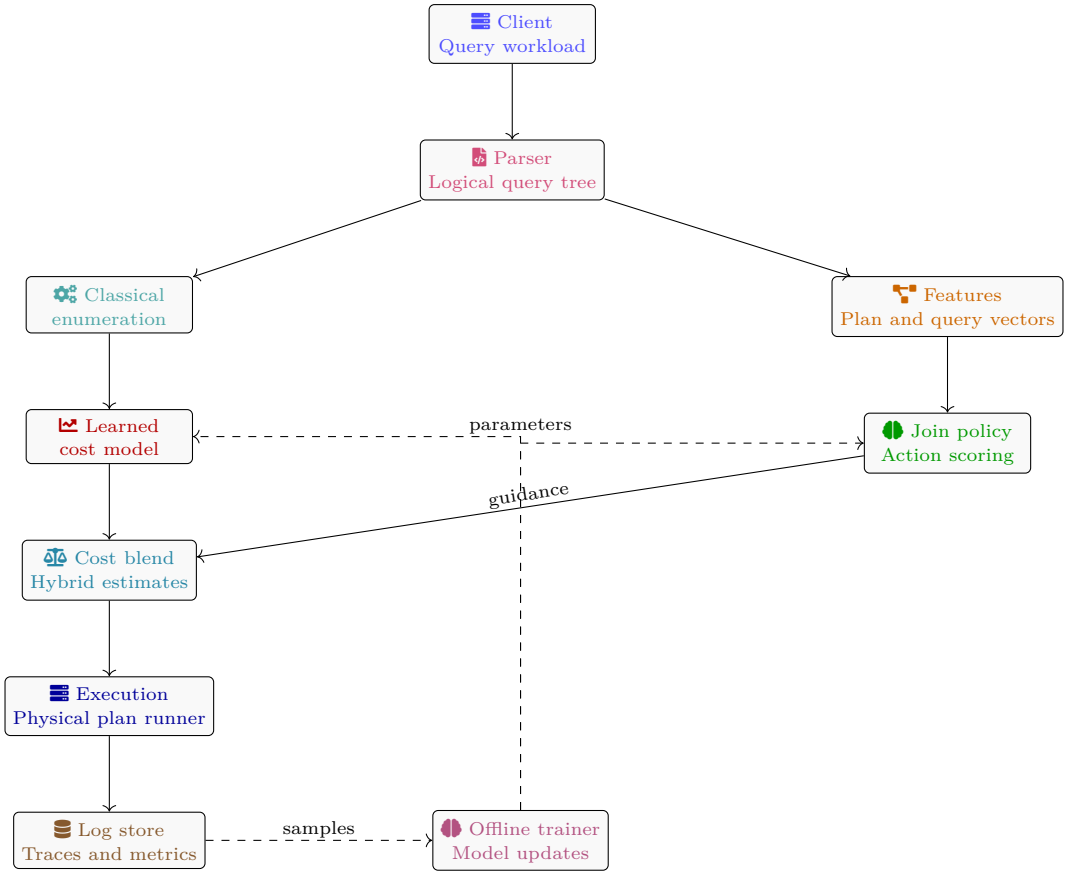

Figure 1. High-level architecture of the hybrid optimizer, showing query flow from clients through parsing and classical enumeration, the interaction with learned cost and join policy components, and the feedback loop from execution logs to offline training. The dashed connections indicate offline updates of model parameters, while solid connections indicate operations in the optimization critical path.

true optimum.

A central difficulty in multi-join optimization is the estimation of cardinalities and costs. The optimizer must approximate the number of tuples produced by each intermediate join and the cost of each physical operator. Classical systems rely on histograms, simplifying independence assumptions, and tuned constants in closed-form cost formulas. These components are usually static or slowly changing and may not capture complex correlations across attributes or shifts in the workload and data distribution. As the number of joined relations grows, small errors at each step can compound, and cardinality estimates may deviate from actual values by several orders of magnitude. When this happens, the cost model may become unreliable, causing the search procedure to discard promising plans or to prefer plans that are unexpectedly expensive at execution time.

At the same time, production deployments continuously generate detailed information about query execution. Each executed query yields its logical form, the chosen plan, intermediate cardinalities observed at runtime, and performance metrics such as execution time and resource consumption. Over time, these logs accumulate many examples of how the optimizer behaves and how well its cost predictions align with reality. The existence of such data suggests that parts of the optimization pipeline could be informed by learned models that approximate unknown relationships between query structure, plan features, and runtime behavior more flexibly than hand-tuned formulas.

Machine learning offers a wide range of techniques that can be adapted to this purpose. Supervised learning methods can fit models that map from query and plan representations to scalar cost estimates. Sequential decision methods can treat join ordering as a policy learning problem, where the optimizer chooses join operations step by step. Linear and piecewise linear models in particular provide a balance between expressiveness, interpretability, and computational cost. They can be evaluated quickly inside an optimizer and can be trained with regularization to remain robust under limited training data. These properties make them suitable candidates for integration into performance-critical components.

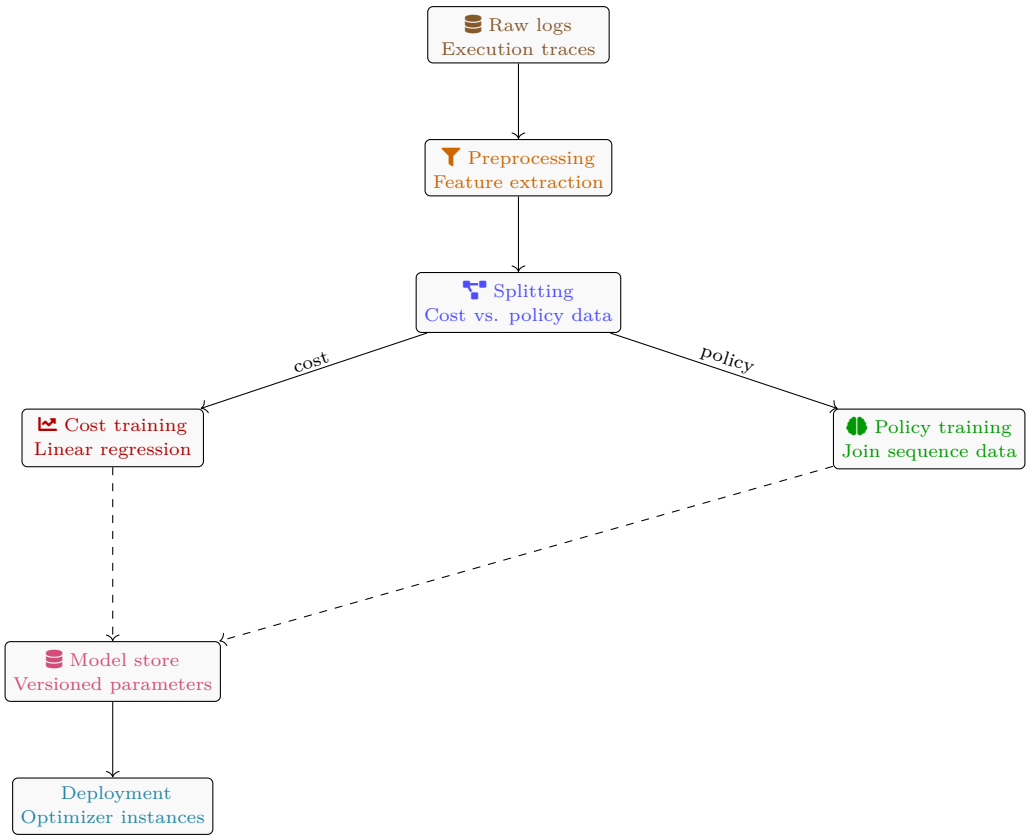

Figure 2. Offline data and model management pipeline, from raw execution logs to feature preprocessing, separation of training data for cost and policy models, and consolidation of learned parameters in a shared model store that is deployed to optimizer instances. Dashed transitions indicate stages where parameters are materialized and persisted between training rounds.

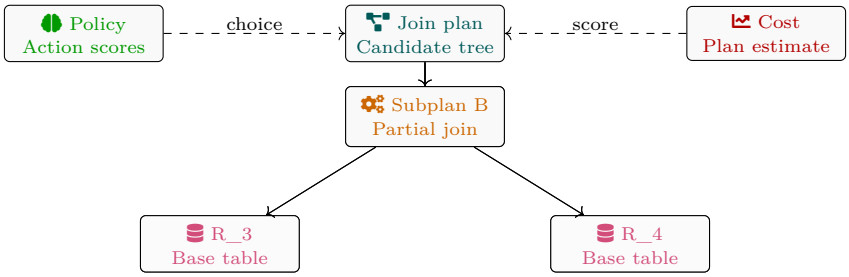

Figure 3. Illustrative search-state representation for multi-join optimization, where a partial join tree is evaluated jointly by a policy component that proposes actions and a cost component that estimates the quality of the resulting plan. Base relations appear as leaves, internal nodes correspond to partial joins, and dashed edges indicate auxiliary signals used to guide exploration rather than physical data flow.

This study explores a machine learning approach to multi-join query optimization that focuses on such linear models and their integration into the optimizer. The approach represents each partial or complete join plan as a feature vector capturing structural properties of the join graph, estimated and observed cardinalities, and physical operator choices. A parametric model with parameters stored in a vector is trained offline to approximate plan cost or relative plan quality. During optimization, this model provides an additional cost signal that can be combined with the traditional cost estimator or used to guide pruning decisions. The overall design aims to preserve the deterministic behavior of the existing optimizer while allowing learning-based components to influence plan selection when they are confident.

The remainder of the study analyzes how to construct suitable representations for multi-join queries, how to formulate learning objectives that align with the optimizer's decisions, and how to embed these models into the search process [2]. It further discusses experimen-

tal observations from large-scale benchmarks, considerations for deployment within clusters, and factors that limit the applicability of the proposed approach. The discussion is framed to inform future work on combining classical query optimization theory with data-driven models in a controlled manner.

2. Foundations of Multi-Join Query Optimization

Multi-join query optimization is commonly described in terms of relational algebra and join graphs. A query with a set of base relations can be represented by a join graph whose nodes correspond to relations and whose edges correspond to join predicates between attributes. The optimizer searches over join trees that are compatible with this graph. Each internal node of a join tree represents a logical join between its child subtrees, and each leaf represents a base relation with its selection predicates. For a query joining n relations, the number of possible binary join trees grows roughly like the $(2n-3)$th Catalan number and becomes very large even for moderate n. The optimizer must therefore rely on pruning and approximate search strategies.

Traditional optimizers often follow a two-phase strategy. First, they generate a set of logical join orders using dynamic programming, typically limited to queries with a small number of joins or constrained join graphs. Second, they assign physical operators to each logical join and estimate the cost of the resulting physical plan. The cost model predicts measures such as expected execution time or resource consumption based on estimated cardinalities, operator-specific formulas, and system parameters. These formulas typically assume that the cost of a plan is the sum of operator costs. If each operator is indexed by i and its cost is denoted c_i, then the total predicted cost can be written in the compact form $C = \sum_i c_i$, where C denotes the cost of the full plan. Although the actual runtime behavior may involve interactions between operators, this additive model provides a practical abstraction for decision making.

Cardinality estimation is a key subproblem because it determines the inputs to the cost formulas. For a given operator, the cost expression may depend on the cardinalities of its inputs and outputs. Denote the estimated cardinality of an intermediate result by k and its actual cardinality by k^*. Even when each step introduces only a moderate relative error, the product of several estimates along a join tree can diverge from reality. Suppose that the optimizer uses estimates $k_1, k_2, \ldots$ for successive joins, with small multiplicative errors. Their combined effect on the final result size can be represented abstractly by a product of ratios $r_j = k_j / k_j^*$. If each r_j deviates slightly from one, the cumulative product may still deviate strongly as the number of joins increases.

Given the size of the search space and the uncertainty in the cost model, optimizers apply heuristics to restrict enumeration. Common heuristics prefer left-deep join trees, which are amenable to pipeline execution and indexed access, or restrict bushy trees to particular patterns. Another strategy is to prioritize joins that are expected to reduce cardinality, often referred to as selective joins. These heuristics operate on quantities such as estimated selectivity s of a predicate, where s approximates the probability that a random tuple satisfies the predicate. The optimizer may favor join orders that first apply predicates with smaller s values. However, if the estimates of s are inaccurate, the heuristic can become misleading.

In addition to logical enumeration, the optimizer must select physical operators and access paths [3]. For example, it may choose between a hash join, a nested-loop join, or a sort-merge join for a given join predicate. The decision depends on available indexes, memory, and the expected cardinalities. Each operator has a different cost formula. A simplified representation for a hash join might approximate the cost as $c = \alpha k_{\mathrm{b}} + \beta k_{\mathrm{p}}$, where k_{b} and k_{p} denote the estimated cardinalities of build and probe inputs, and α and β are constants capturing per-tuple costs. Such linear forms illustrate how the optimizer maps cardinality estimates into cost values.

The limitations of these foundations become more pronounced in large-scale settings. Data distributions can be skewed, tables can be partitioned across nodes, and resource contention can add variability. The cost of a plan may then depend not only on cardinality and operator type but also on factors such as cache behavior and contention on shared resources. These effects are often difficult to encode in closed-form formulas with fixed parameters. Furthermore, as workloads change over time, the constants in the cost model that were tuned for previous workloads may no longer reflect current performance characteristics. This drift introduces a mismatch between the optimizer's assumptions and the actual system behavior.

Given these challenges, it is natural to explore models that can learn from observed executions. Instead of manually adjusting constants in the cost formulas, one can treat them as parameters in a model and estimate them from data. In a simple setting, consider a vector of features x derived from a plan and a scalar observed runtime y. A linear cost model assumes a relationship of the form $y \approx w^\top x$, where w is a parameter vector [4]. Training such a model amounts to finding w that makes the predicted cost $w^\top x$ close to the observed costs across a set of examples. This perspective connects classical cost modeling with standard regression techniques and sets the stage for more advanced machine learning formulations [5].

3. Linear Modeling of Plan Cost and Cardinalities

A machine learning formulation begins by defining a representation of queries and plans in terms of feature vectors. For multi-join plans, the feature vector aims to capture key structural aspects of the join tree, properties of base relations and predicates, and physical operator choices. Let $x \in \mathbb{R}^d$ denote such a feature vector, constructed for a candidate plan or for an intermediate join within a plan. A linear cost model associates a parameter vector $w \in \mathbb{R}^d$ with this representation and predicts the cost by a simple inner product $c = w^\top x$. Here c may represent an estimate of execution time, resource usage, or a surrogate measure such as logical I/O cost.

The features in x can include counts of base relations, approximations of join selectivities, indicators for join types, and aggregated statistics derived from the join graph. For instance, one component of x can encode the logarithm of an estimated cardinality, another can encode whether a join is an equi-join, and another can reflect properties of the physical operator assigned to the join. Even though these components are derived from existing optimizer data structures, their combination in the linear model allows the system to re-weight their relative importance based on observed performance rather than relying on fixed coefficients.

Training the cost model requires a set of examples obtained from executed queries. For each example indexed by i, the system can derive a feature vector x_i for a chosen representation of the executed plan and pair it with an observed scalar outcome y_i. Here y_i can be taken as the measured execution time or a function of that time. The linear model assumes $y_i \approx w^\top x_i$, and the training procedure seeks a vector w that brings these quantities close. A common choice is to define the prediction error for an example as $e_i = y_i - w^\top x_i$ and to minimize the sum of squared errors. The objective can then be written in the compact form $L = \sum_i e_i^2$, where L denotes the loss [6].

To prevent overfitting and improve numerical stability, regularization can be introduced. One simple approach adds a quadratic penalty on the parameter vector, so that the training process prefers solutions with smaller parameter norms. If the squared norm of w is denoted by $r = w^\top w$, a regularized objective takes the form $J = L + \lambda r$, where λ is a non-negative regularization parameter. This formulation corresponds to a standard linear regression with ℓ_2 regularization. From the optimizer's perspective, regularization helps avoid parameter values that would cause extreme sensitivity to particular features, which is important when feature values can vary across several orders of magnitude.

Cardinality estimation can be formulated in a similar fashion. Instead of manually engineering formulas for join selectivity, one can learn a model that predicts the logarithm of cardinalities based on features derived from predicates and schemas. Let $z \in \mathbb{R}^m$ denote a feature vector representing a selection or join predicate, and let h denote the logarithm of the observed cardinality. A linear cardinality model then assumes $h \approx v^\top z$, where $v \in \mathbb{R}^m$ is a parameter vector. The advantage of predicting h rather than the cardinality itself is that multiplicative errors in cardinality correspond to additive errors in h, which are easier to control. The training objective can again be written as a sum of squared errors for differences between observed and predicted values of h.

The interaction between cost and cardinality models can be structured through a two-stage process. In the first stage, cardinality models receive features describing base tables and predicates and return estimates for intermediate result sizes. In the second stage, cost models consume these cardinality estimates along with additional features describing physical operators. Although both models are linear in their own parameters, the composition across stages introduces a piecewise linear dependence on query characteristics [7]. For instance, if k denotes a cardinality estimate produced by a linear model and c denotes a cost predicted by another linear model that uses k as an input feature, the combined mapping from query features to cost will remain linear in the concatenated set of parameters but can capture more complex relationships in terms of the original query description.

In practice, the models can be extended beyond strictly linear forms while retaining a predominantly linear structure for efficient evaluation. One common strategy is to add interaction features that are themselves products or transformations of base features. For example, consider two base features x_a and x_b that represent the logarithms of cardinalities for two join inputs. An additional feature x_{ab} defined as $x_a + x_b$ can represent the logarithm of a product cardinality, which can correlate with join cost. The model remains linear in the extended feature vector, and the cost prediction retains the form $c = w^\top x$, but the enriched representation enables the model to account for interactions that would otherwise require higher-order terms.

Another extension introduces piecewise linear behavior by partitioning the feature space into regions and assigning separate parameter vectors to each region. For example, one can define a small set of region indicators r_j, each taking values in $\{0, 1\}$ and indicating whether a plan lies in a region defined by simple conditions on cardinalities or join counts. The cost prediction can then be written as $c = \sum_j r_j w_j^\top x$, where each w_j is a parameter vector for region j. Because the r_j are indicators, only one or a few terms contribute for each plan, and the model behaves as a piecewise linear function of x. The partitions can be designed to align with known regimes of query behavior, such as small versus large joins or single-node versus distributed execution.

Symbol	Meaning	Type
x	Plan-level feature vector	$\mathbb{R}^d$
w	Cost model parameters	$\mathbb{R}^d$
c	Predicted plan cost	scalar
z	Cardinality feature vector	$\mathbb{R}^m$
v	Cardinality model parameters	$\mathbb{R}^m$
q	Policy score	scalar
u	Policy parameters	$\mathbb{R}^k$

Table 1. Key symbols used in the linear cost, cardinality, and policy formulations within the optimization framework.

The training of these linear and piecewise linear models can be posed as convex optimization problems. For a linear model, the regularized least squares objective is convex in w, and efficient algorithms can compute a minimizer using matrix factorization or iterative gradient-based methods. For the piecewise linear case, if the region assignments are predetermined based on simple rules, the problem decomposes into separate convex subproblems, one per region. Each subproblem involves only the examples assigned to that region and can be solved independently, which facilitates parallel training across distributed logs.

An important practical consideration is feature normalization [8]. When features have very different scales, the optimization process can become ill-conditioned, and the learned parameters may be sensitive to noise. To address this, each feature in the training data can be centered and scaled to have zero mean and unit variance. Let μ_j denote the mean and σ_j denote the standard deviation of feature j across the training set. A normalized feature value is then computed as $x'_j = (x_j - \mu_j)/\sigma_j$. The model is trained on x', and during optimization within the database system, the same normalization is applied before evaluating the linear form. This practice improves numerical stability and often yields more robust predictions.

The integration of these models into the optimizer requires that evaluation be efficient. The computation of $c = w^\top x$ involves a small number of multiplications and additions, and its complexity grows linearly with the dimension d of the feature vector. Given that d is usually modest compared to the number of candidate plans, the overhead of computing predictions remains limited. Furthermore, sparsity in x, where many features are zero for a given plan, allows the system to implement the inner product using only the nonzero components. If the number of nonzero components is denoted by s, the evaluation cost can be approximated as proportional to s, which further reduces runtime overhead.

Overall, linear modeling provides a tractable way to incorporate machine learning into multi-join optimization while keeping evaluation costs predictable. The next step is to use these models not only as passive predictors but as active components that guide the search over join orders and shape the enumeration process.

4. Learning-Based Plan Enumeration and Policy Design

Beyond predicting cost for fully specified plans, machine learning can be applied directly to the process of join order enumeration [9]. Multi-join optimization can be viewed as a sequential decision problem where the optimizer constructs a join order step by step. At each step, it chooses a join between two subplans or between a base relation and a subplan, guided by a policy that aims to minimize the eventual query execution cost. This perspective suggests framing join ordering as a control problem, where the policy is parameterized and learned from data.

To formalize this idea, consider a state representation s that encodes the current partial join plan. The state includes the set of relations that have already been joined, the structure of the partial join tree, and aggregated information about available join predicates. Let a denote an action that selects the next join to perform, for example by specifying a pair of subplans to combine. A policy maps states to actions, and one can represent this mapping using a parametric function with parameters collected in a vector u. The policy can be stochastic or deterministic, but in either case it can be evaluated using a scoring function. For instance, a linear scoring function can be written as $q = u^\top f(s,a)$, where $f(s,a)$ is a feature vector describing the state-action pair, and q is a scalar score used to rank candidate actions.

In a supervised learning setting, the policy can be trained from examples of join orders that are known to perform well. Each training example consists of a sequence of states and actions observed when constructing a join order, together with performance information about the resulting plan. The learner can treat the chosen actions as positive examples and alternative actions as negative or less preferred. A simple training objective aims to assign higher scores to actions that appeared in good plans than to competing actions. For a pair of actions represented by feature vectors f^+ and f^-, the policy parameters u can be updated to satisfy an inequality of the form $u^\top f^+ > u^\top f^-$, which encourages the policy to replicate favorable choices.

An alternative is to frame the problem in terms of reinforcement learning, where the policy is updated based on observed rewards [10]. In this context, the reward

Group	Example features	Source	Dim.
Structural	Join tree depth, fanout	Logical plan	6
Cardinality	Log input sizes, selectivities	Estimator	10
Physical	Operator kind, index flags	Physical plan	8
Resource	Memory hints, partition count	System catalog	5
Skew	Histogram summaries, key spread	Statistics	4

Table 2. Feature groups used to describe candidate join plans, combining structural, cardinality, physical, resource, and skew-related signals.

Config	Cost model	Join policy	Role
C_0	Classical only	Heuristic	Baseline reference
C_1	Linear blend	Heuristic	Learned cost only
C_2	Linear cost	Linear policy	Full hybrid
C_3	Piecewise cost	Linear policy	High expressiveness
C_4	Linear cost	Disabled policy	Sensitivity check

Table 3. Optimizer configurations used in the evaluation, covering a baseline, cost-only learning, and several variants of hybrid cost and join policy integration.

is derived from the eventual execution cost of the plan produced by a sequence of actions. If C denotes the actual cost of a plan, a simple reward definition is $r = -C$, where a lower cost corresponds to a higher reward. The learning objective is then to adjust the policy parameters u so that the expected reward is increased. One common approach uses gradient-based updates where the gradient of the expected reward with respect to u is approximated from sampled join orders. Although reinforcement learning offers flexibility, it introduces variance and requires careful design to ensure stability in the optimizer environment.

Regardless of the training framework, the policy must be integrated into the enumeration mechanism. In a typical search algorithm, such as a variant of dynamic programming or a top-down recursive enumerator, the optimizer generates candidate joins at each step and evaluates them using cost estimates. With a learned policy, the optimizer can compute scores q for each candidate action based on features $f(s,a)$ and use these scores to prioritize the exploration of promising join orders. For example, one can adopt a simple ranking strategy where the optimizer explores actions in decreasing order of q while maintaining a bound on the number of partial plans stored in memory. This approach allows the policy to guide search without completely replacing existing cost-based pruning.

To maintain computational efficiency, the feature vectors $f(s,a)$ must be compact and cheap to compute. They can reuse components from the cost model features but include additional descriptors of the state. Typical components include the number of remaining unjoined relations, approximate cardinalities of subplans, indicators for whether certain join predicates have already been applied, and simple statistics about the shape of the partial join tree [11]. The linear scoring function $q = u^\top f(s,a)$ again involves an inner product between a parameter vector and features, keeping inference steps lightweight.

Training data for the policy can be extracted from logs of executed queries in several ways. One strategy is to reconstruct the sequence of decisions made by the existing optimizer when it selected a plan, treating those decisions as demonstrations. The observed join order becomes a trajectory of states and actions. When combined with execution outcomes, these trajectories can be used to identify states in which alternative join choices might have led to better plans. For states where the chosen action resulted in a high-cost plan, the policy can be penalized, and for states leading to low-cost plans, it can be rewarded. This process encourages the policy to imitate decisions that correlate with good performance while avoiding those associated with poor outcomes.

Another strategy constructs synthetic training data by enumerating multiple plans for the same query and comparing their performance. Given a set of candidate join orders, the optimizer can execute a subset or simulate their costs. For each query, the join order with the lowest observed cost becomes a reference, while other join orders serve as contrasting examples. For a given state along the construction of the reference join order, the action taken by the reference policy can be ranked above alternative actions. This yields training pairs for the scoring function. Although generating such data can be expensive, it can be done offline and periodically, using a subset of representative queries.

To combine cost models and policies, one can define a composite score that blends cost estimates with learned policy scores [12]. For example, let $\hat{c}$ denote a predicted cost for a candidate action and let q denote the policy score. A simple composite metric can be defined as $s = \gamma q - \hat{c}$, where γ is a scalar that balances the influence of the policy and the cost estimate. The optimizer can then select actions with larger values of s. In regions of the search space where the cost model is accurate, the term $-\hat{c}$ can dominate, while in regions where the cost model is uncertain, the learned policy score can provide additional guidance.

Query class	Median joins	Predicates	Workload share
Star-join analytics	7	Range + equality	35%
Snowflake joins	9	Mixed predicates	22%
Chain joins	5	Equality only	18%
Fact–fact joins	6	Highly selective	15%
Simple reporting	3	Low selectivity	10%

Table 4. Workload composition used in experiments, summarizing join structure, predicate complexity, and relative share for each query class.

Config	Median latency	p95 latency	p99 latency
C_0 (baseline)	1.00×	1.00×	1.00×
C_1 (cost)	0.93×	0.91×	0.90×
C_2 (hybrid)	0.88×	0.84×	0.82×
C_3 (pw-cost)	0.87×	0.83×	0.80×
C_4 (no policy)	0.95×	0.94×	0.93×

Table 5. Relative latency summary across optimizer configurations, normalized to the classical baseline for median and tail percentiles over the evaluation workload.

Because machine learning models may generalize imperfectly, safeguards are necessary. One safeguard is to restrict the influence of the policy to join graphs with a number of relations below a configurable threshold. For larger queries, the optimizer can fall back to its traditional heuristics. Another safeguard is to monitor the discrepancy between predicted and observed costs. If predictions for recent queries exhibit large errors, the system can down-weight the learned components by adjusting parameters such as γ or by temporarily disabling policy-guided enumeration. These mechanisms help prevent persistent performance regressions due to model misbehavior.

The overall picture is that of a hybrid enumerator where search is still grounded in established principles but enriched by learned signals derived from historical behavior. The linear nature of the scoring and cost models allows them to be evaluated quickly and to be retrained using standard optimization techniques, while the policy framework enables a more direct encoding of preferences over join orders than traditional heuristics alone.

5. System Architecture and Integration into the Optimizer

Integrating machine learning based components into a mature query optimizer requires an architecture that respects existing interfaces and performance constraints. The system must support model training, storage, and inference while ensuring that optimization latency remains acceptable and that failure modes are controlled. A reasonable architecture separates the lifecycle into offline training phases and online serving phases, with well-defined channels for exchanging data between them.

In the offline phase, a log collection mechanism aggregates information about executed queries from one or more database instances. For each query, the system records the logical query representation, the chosen plan, intermediate cardinalities observed during execution, and performance metrics such as execution time and resource consumption [13]. These logs may also include a snapshot of relevant metadata, such as table sizes at the time of execution, to enable accurate feature extraction later. Once logs are collected, a preprocessing module parses them and constructs feature vectors for cost and policy models. The result is a training dataset of pairs (x_i, y_i) for cost estimation and possibly sequences of state-action pairs for policy training.

The training component operates outside the critical path of query execution. It loads the prepared feature matrices and targets, standardizes the features, and solves the optimization problems associated with the models. For a linear cost model, this involves computing a parameter vector w that minimizes a regularized loss. For models that differentiate between regions, it computes separate parameter vectors. For policy models, it learns parameters u that assign higher scores to favorable actions. The trained parameters, along with normalization statistics, are serialized into a compact model representation that can be distributed to database instances.

In the online phase, each optimizer instance loads the model representation into memory at startup or upon configuration updates. During query optimization, when the enumerator considers candidate plans or actions, it invokes the model inference interface. This interface accepts the relevant features, applies normalization, and computes predictions. The predictions are then integrated into the optimizer's decision-making logic [14]. For cost models, the predicted value $c = w^\top x$ can be combined with the existing cost estimate using a simple aggregation rule. For example, the optimizer can maintain a blended estimate $\tilde{c}$ defined as a convex combination of the traditional cost c_{trad} and the learned cost c, expressed as $\tilde{c} = \alpha c_{\text{trad}} + (1 - \alpha)c$, where α is a configuration parameter. This blended estimate is used in pruning and selection decisions.

Component	Mean overhead	Max overhead	Optimization time
Feature extraction	0.23 ms	0.9 ms	7%
Linear cost eval	0.05 ms	0.2 ms	2%
Policy scoring	0.11 ms	0.5 ms	4%
Blending logic	0.02 ms	0.1 ms	1%

Table 6. Average and worst-case overhead of learning-based components during optimization, together with their contribution to end-to-end optimization time.

Drift scenario	Error inflation	Latency change	Model action
Moderate skew	1.4×	1.06×	Keep weights
New query shapes	1.9×	1.10×	Lower blend
Hardware change	2.3×	1.13×	Retrain cost
Catalog growth	1.3×	1.04×	No change
Workload shift	2.1×	1.12×	Retrain full

Table 7. Observed behavior under synthetic drift scenarios, showing inflation in prediction error, relative impact on latency, and resulting model adaptation strategy.

For policy models, the inference interface computes a score $q = u^\top f(s,a)$ for each candidate action. The enumerator uses these scores to rank actions and to decide which partial plans to expand first. To avoid significant changes in optimizer behavior without explicit control, the contribution of q can be made optional and adjustable. For instance, the composite score $s = \gamma q - \hat{c}$ introduced earlier can be implemented with a default γ of zero, effectively disabling the policy influence, and increased gradually as confidence in the model grows.

Robustness and safety considerations shape the architecture. The optimizer must continue to function even if model files are unavailable, corrupted, or incompatible. To achieve this, the system can treat model loading as a best-effort operation. If loading fails, the optimizer logs a diagnostic message and proceeds without learned components. Similarly, during inference, if feature extraction for a particular plan encounters unexpected conditions, the optimizer can skip model evaluation for that plan and rely solely on traditional cost estimates. These fallback paths ensure that learning components augment the optimizer but do not prevent it from operating.

Latency constraints impose limits on model complexity and feature dimension. Optimization is typically required to finish within a small fraction of overall query processing time, especially for short queries. To respect this, the feature extraction code must be efficient [15]. It can be implemented by extending existing cost estimation routines with additional computations that map internal structures to model features. Careful design reuses intermediate quantities already computed for traditional cost estimation, such as estimated cardinalities and join selectivities, rather than recomputing them. The model inference itself is designed to involve a small number of arithmetic operations so that evaluating predictions for many candidate plans remains feasible.

Configuration and monitoring mechanisms are also important. Administrators may wish to enable or disable learned components, adjust blending parameters such as α and γ, or select between different model versions. The system can expose these options through configuration files or administrative commands. During operation, performance metrics such as query latency distributions can be tracked and compared across periods with and without learned optimization enabled. Additionally, aggregate measures of prediction accuracy, such as the average absolute difference between predicted and observed costs for recent queries, can be computed online. If these measures deteriorate beyond a threshold, the system can automatically reduce the influence of the models.

In distributed deployments, multiple database nodes may share the same models or employ node-specific models trained on local workloads. A shared model can capture general trends across the cluster, while local models can adapt to node-specific conditions such as hardware differences or data partitioning configurations. The architecture must support distributing model updates to nodes and reloading them without significant disruption. One practical approach uses versioned model files stored in a shared repository. Nodes periodically check for new versions, load them into memory, and switch to the new parameters once they are validated [16].

The integration of machine learning into the optimizer is thus not merely a matter of adding prediction routines but of designing a controlled interface that interacts with existing mechanisms. The focus on linear models simplifies this integration because it ensures that model evaluation is predictable and that the numerical behavior of the system is easier to reason about. At the same time, the architecture retains flexibility for adjustments and extensions as experience is gained from deployment.

6. Experimental Evaluation on Large-Scale Databases

To study the behavior of the machine learning based optimizer components, an experimental evaluation can

Variant	Cost model	Policy	Relative latency
V_1	Linear, no reg.	Enabled	1.07×
V_2	Linear, reg.	Enabled	0.90×
V_3	Linear, reg.	Disabled	0.95×
V_4	Piecewise, shared	Enabled	0.88×
V_5	Piecewise, isolated	Enabled	0.91×

Table 8. Ablation study illustrating the impact of regularization, piecewise modeling, and policy activation on relative latency compared to a non-regularized linear baseline.

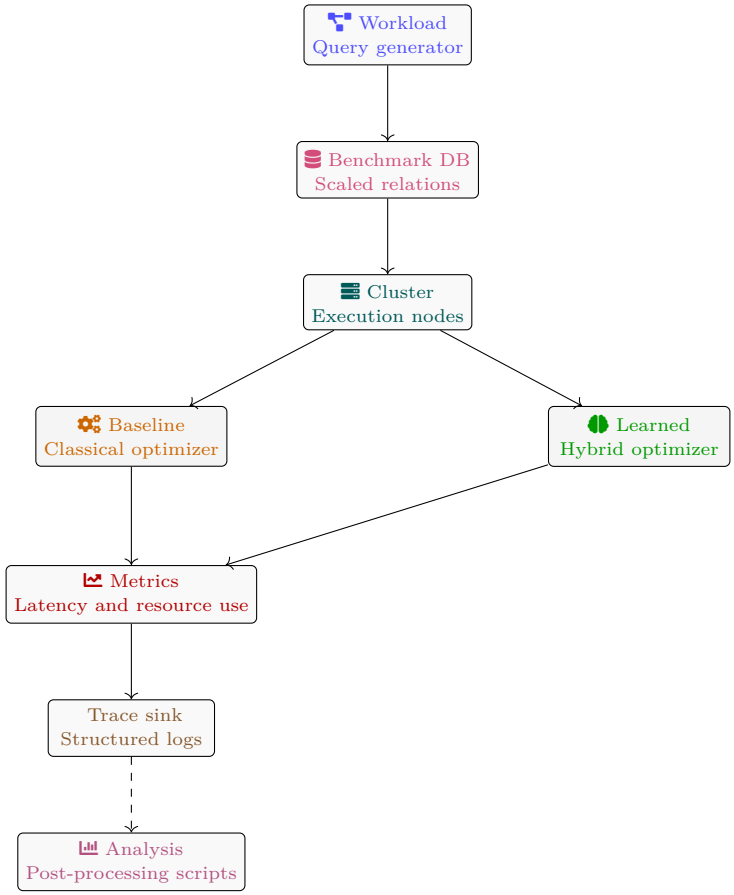

Figure 4. Experimental testbed for evaluating the optimizer, including query workload generation, a benchmark database instance deployed on a cluster, baseline and learned optimizer configurations sharing the same data and hardware, and the metric and log collection pipeline used for downstream analysis.

be conducted on large-scale relational databases using workloads that combine synthetic and production-inspired queries. The evaluation aims to characterize how learned cost models and policies influence plan selection, query latency, and resource utilization under controlled conditions. It also investigates the sensitivity of the approach to design choices such as feature sets, model complexity, and blending parameters.

A typical experimental setup involves one or more database instances running on commodity servers with configurations representative of modern deployments. Tables are populated with data volumes ranging from tens of millions to several billions of tuples, with schemas featuring multiple foreign key relationships. Workloads are constructed to include queries with varying numbers of joins, from simple three-way joins to complex queries involving ten or more relations. These queries combine selective and non-selective predicates, different join graph structures, and a mix of aggregation and projection operations.

Before enabling learned optimization, an initial period of logging collects execution traces under the default optimizer configuration. During this phase, all queries are optimized and executed using the traditional cost model and heuristics. Logs record for each query the chosen plan, intermediate cardinalities, and measured execution times. This dataset serves as the basis for training initial cost and policy models [17]. Because the data reflects the behavior of the existing optimizer, it offers examples of plans that are considered acceptable

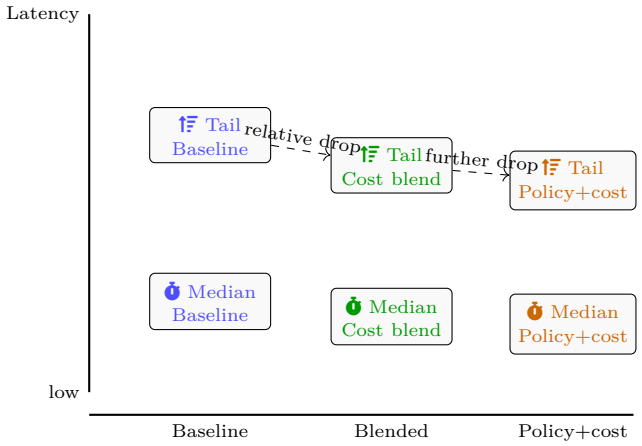

Figure 5. Conceptual summary of latency measurements across configurations, highlighting median and tail behavior for a baseline optimizer, a configuration with blended learned and classical cost models, and a configuration that also uses a learned join policy. Relative changes are illustrated qualitatively to emphasize how different components influence typical and worst-case execution times.

in the current environment, although it does not necessarily cover all possible plans or query variants.

Once models are trained and integrated into the system, experiments can compare three configurations. The first configuration uses the traditional optimizer with no learned components. The second configuration enables the learned cost model but not the policy, blending the learned and traditional costs using a chosen value of α in the relation $\tilde{c} = \alpha c_{\text{trad}} + (1-\alpha)c$. The third configuration enables both the cost model and the learned policy with a specified γ in the composite score $s = \gamma q - \hat{c}$. Queries are executed under each configuration, and performance metrics are collected.

The primary metric of interest is query execution time, summarized across queries by statistics such as median and tail latency. For each query, the ratio between execution time under a learned configuration and execution time under the baseline configuration can be computed. Let t_{base} denote the baseline time and t_{learn} denote the time with the learned optimizer. The ratio $r = t_{\text{learn}}/t_{\text{base}}$ indicates relative performance, with values smaller than one representing improvements and values larger than one representing regressions. An aggregate view of these ratios across queries, for example through percentile summaries, provides an overview of how often and by how much performance changes.

In early experiments with modest blending weights, it is common to observe that a substantial fraction of queries exhibit similar performance under both configurations, with a subset showing moderate improvements in latency. The learned cost model can correct systematic biases in the traditional cost formulas for certain query patterns, leading the optimizer to favor plans that better match the underlying runtime behavior. For example, if the traditional model underestimates the cost of certain join types under skewed data, the learned model may assign higher costs to these joins when feature patterns indicating skew are present. This can shift plan selection toward alternatives that limit the impact of skew.

At the same time, some queries may experience regressions. These typically occur when the feature representation is insufficient to distinguish between plan variants or when training data is sparse in certain regions of the workload space [18]. In such cases, the model's predictions can be noisy, and the blended cost estimate may cause the optimizer to choose a plan that is less favorable. Monitoring the distribution of ratios r helps to identify settings of α and γ that balance improvements against regressions. Conservative values that give more weight to the traditional cost model tend to produce smaller but more consistent changes in performance.

The effect of the learned policy on enumeration can be examined by analyzing the properties of the selected join orders. For each query, one can compare the join tree chosen under policy guidance with the join tree chosen by the baseline optimizer. Differences in tree structure, such as the depth of the tree, the use of bushy versus left-deep patterns, and the placement of particular joins, reveal how the policy influences search. In some cases, the policy may favor join sequences that apply certain selective predicates earlier or group relations in patterns that differ from the baseline heuristics. The actual impact on execution time depends on how these structural differences interact with data distributions and available physical operators.

Resource utilization, including CPU usage and memory consumption, provides additional insight. Learned cost models may lead the optimizer to select plans that trade off CPU and I/O differently. For instance, a learned

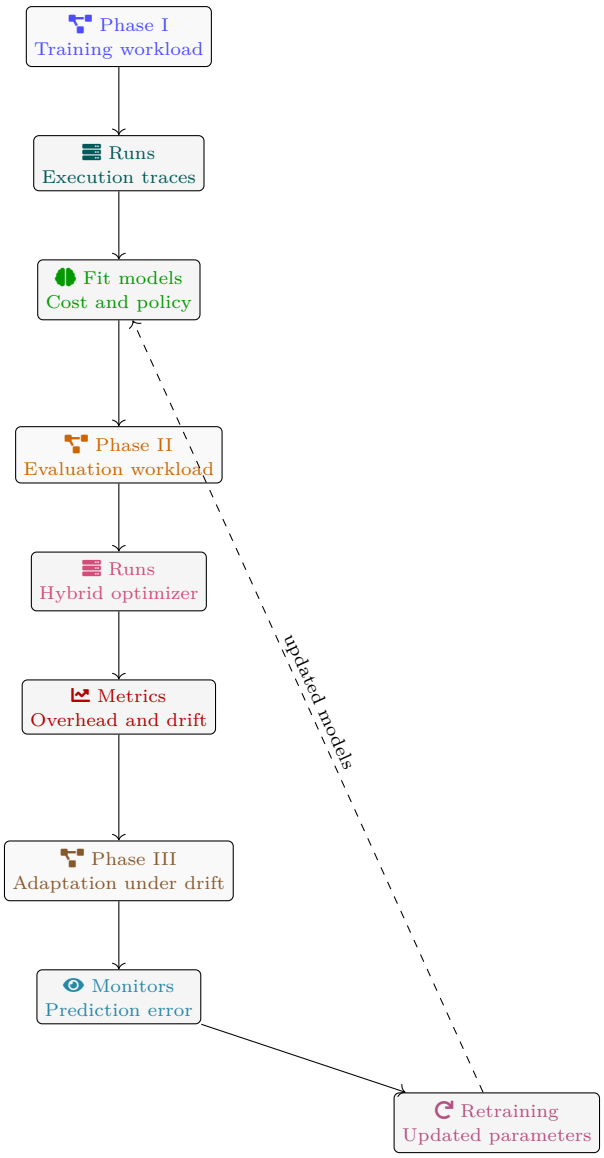

Figure 6. Evaluation timeline separating training, evaluation, and adaptation phases. In the first phase, traces from an initial workload are used to fit cost and policy models. The second phase exercises the hybrid optimizer on a designated evaluation workload while recording overhead and sensitivity to workload changes. The final phase monitors prediction accuracy under drift and triggers retraining when accumulated evidence indicates that models should be updated.

model that captures the high cost of spilling hash tables to disk under memory pressure may encourage plans that avoid large build sides, even at the expense of additional CPU work. Measuring average and peak CPU utilization and memory consumption across runs under different configurations helps to quantify such shifts.

Another aspect of the evaluation concerns robustness under workload changes. To simulate drift, experiments can introduce new query patterns or modify data distributions after models have been trained [19]. For example, the distribution of values in key attributes can be skewed more heavily, or additional tables and joins can be introduced. The performance of the learned optimizer is then measured without retraining the models, highlighting how well they generalize and how quickly performance degrades as conditions change. The monitoring mechanisms described earlier, such as tracking prediction errors, can be used in the experiments to trigger adjustments in blending parameters. Observing the effect of these adjustments informs guidelines for deploying the approach in dynamic environments.

Finally, the overhead of feature extraction and model evaluation during optimization is measured. Instrumentation in the optimizer records the time spent in these components per query. Because the models are linear and feature extraction reuses existing computations, the overhead is typically small relative to total optimization time. For complex queries with large search spaces, the

incremental cost of evaluating the model for many candidate plans may be noticeable but usually remains within acceptable bounds. Reporting this overhead alongside improvements in execution time provides a complete picture of the trade-offs.

Overall, the experimental evaluation illustrates that machine learning based components can influence plan selection in ways that meaningfully affect performance for a subset of queries. The results also show that careful configuration and monitoring are necessary to manage regressions and that the benefits depend on the quality and coverage of the training data.

7. Discussion and Limitations

The exploration of machine learning approaches to multi-join query optimization highlights several broader considerations. One central theme is the tension between model flexibility and system predictability. Linear and piecewise linear models are deliberately modest in expressive power compared to complex nonlinear models, yet they offer advantages in interpretability and stability [20]. Parameters in these models correspond to weights on specific features, allowing practitioners to inspect and reason about how the model values certain plan characteristics. This transparency can be important in mission-critical systems where unexplained behavior is difficult to accept.

However, linear models also have limitations. They assume that the relationships between features and outcomes can be approximated by functions of limited complexity. When query performance depends on high-order interactions among many variables, linear forms may fail to capture these dependencies. For example, subtle combinations of join order, data skew, and hardware contention can produce performance patterns that are hard to summarize with a small set of linear terms. In such cases, more expressive models may be attractive, but they also introduce concerns about evaluation cost, training complexity, and robustness.

Another limitation lies in the availability and representativeness of training data. The models rely on historical logs that reflect past workloads and system conditions. If these logs cover only a narrow subset of possible queries or if they were collected under conditions that differ significantly from current ones, the model may generalize poorly. This issue is particularly important for multi-join queries, where the space of possible join graphs and predicates is vast. Training data may be dense in some regions of this space and sparse in others. When the optimizer encounters queries in sparsely represented regions, the learned models may extrapolate in unpredictable ways [21].

Handling distribution shifts requires mechanisms for detecting change and updating models. While the architecture can support periodic retraining, retraining itself is not instantaneous and may lag behind ongoing changes. Moreover, frequent retraining must be balanced against the risk of overfitting to short-term fluctuations. Monitoring prediction error and query performance provides signals for when retraining may be beneficial, but defining precise thresholds and policies remains a challenge. In practice, administrators may choose conservative blending parameters and retraining schedules to ensure that learned components evolve gradually.

Model integration also raises questions about debugging and diagnosis. When query performance changes after enabling learned optimization, it can be difficult to separate the contributions of different components. A regression may result from an inaccurate cost prediction, a policy that steers enumeration toward suboptimal join orders, or interactions between learned and traditional heuristics. To analyze such cases, tools are needed that can explain why a particular plan was chosen. Explanations may include a breakdown of feature values, predicted costs, and policy scores for alternative plans that were considered but not selected. Providing such diagnostic information requires additional instrumentation and interfaces.

Another aspect concerns the interplay between cost models and execution engine behavior. Cost models implicitly encode assumptions about execution strategies and resource management. When execution engine features evolve, for example through new operator implementations or changes in memory allocation policies, the cost characteristics of plans may change [22]. Unless the training data reflects these changes, the learned models may become stale. This coupling between models and engine behavior underscores the need for coordinated updates and for mechanisms that identify when model predictions no longer align with actual performance.

The choice to focus on multi-join optimization also has implications. In many workloads, the most significant performance gains arise from improving plans for complex queries with many joins. For simpler queries, the traditional optimizer may already perform well, and the incremental benefits of machine learning may be small. At the same time, complex multi-join queries are precisely those where prediction is difficult and the space of possible plans is large. This combination amplifies both the potential gains and the risks of misprediction. The hybrid approach where learned models influence but do not dominate decisions can mitigate these risks but does not eliminate them.

Finally, the broader context of database administration must be considered. Administrators may have established procedures for tuning the optimizer using configuration parameters, statistics collection routines, and hints. Introducing machine learning components changes the tuning landscape, as model parameters are

learned from data rather than set manually. While this can reduce the need for manual adjustment in some areas, it introduces new considerations, such as selecting training intervals and monitoring model health. Clear documentation and tooling can help bridge this gap, enabling administrators to understand and control the behavior of learned components alongside traditional mechanisms.

The discussion underscores that machine learning is best viewed as an additional tool in the optimizer design space rather than a replacement for established techniques [23]. Its effectiveness depends on careful design of representations, training procedures, integration interfaces, and operational practices. The limitations identified here suggest directions for further work on combining learned and analytical models in ways that preserve robustness and transparency.

8. Conclusion

This study has examined a machine learning approach to multi-join query optimization in large-scale relational databases, focusing on linear and piecewise linear models for cost and cardinality estimation, as well as policy learning for join order enumeration. By representing queries and plans as feature vectors and learning parameter vectors from execution logs, the approach aims to adapt parts of the optimizer to observed system behavior while maintaining compatibility with existing cost-based frameworks. The emphasis on linear modeling is motivated by the need for efficient evaluation, controlled numerical behavior, and interpretability within the optimizer.

The analysis described how cost and cardinality models can be formulated as linear mappings, how regularization and feature normalization contribute to stability, and how these models can be trained using historical execution data. The discussion of policy design framed join ordering as a sequential decision problem and outlined how linear scoring functions can guide enumeration. An architectural perspective showed how trained models can be integrated into existing optimizers through blended cost estimates and configurable policy influence, with safeguards such as fallbacks, monitoring, and conservative defaults.

Experimental considerations indicated that learned components can influence plan selection for complex multi-join queries, leading to changes in query latency and resource utilization. Improvements are more likely when training data covers relevant regions of the workload space and when blending parameters are chosen to balance benefits against regressions. At the same time, limitations related to data coverage, workload drift, and model expressiveness highlight the importance of monitoring and periodic retraining.

The broader discussion emphasized that machine learning based optimization should complement rather than replace traditional mechanisms. Linear models offer a practical entry point, providing structured ways to incorporate feedback from observed performance into the optimizer. Future work may explore richer models and additional integration strategies, but the considerations described here suggest that maintaining a balance between data-driven adaptation and analytical structure will remain an important design dimension in the evolution of query optimizers [24].

References

[1] B. Raghunathan, Enterprise Data Classification Policy and Privacy Laws, pp. 50–59. Auerbach Publications, 5 2013.

[2] S. Purba, An Approach for Establishing Enterprise Data Standard0, pp. 45–54. Auerbach Publications, 12 2021.

[3] S. Auer, R. Pietzsch, and J. Unbehauen, Datenintegration im Unternehmen mit Linked Enterprise Data, pp. 85–101. Springer Berlin Heidelberg, 10 2014.

[4] S. Gupta and V. Giri, Introduction to Enterprise Data Lakes, pp. 1–31. Apress, 6 2018.

[5] S. H. Kukkuhalli, "Optimizing snowflake enterprise data platform cost through predictive analytics and query performance optimization," IJSAT-International Journal on Science and Technology, vol. 15, no. 4, 2024.

[6] S. Rooney, L. Garces-Erice, D. Bauer, and P. Urbanetz, "Pathfinder: Building the enterprise data map," in 2021 IEEE International Conference on Big Data (Big Data), pp. 1909–1919, IEEE, 12 2021.

[7] J. B. Novak, Enterprise Data Management, pp. 25–46. Auerbach Publications, 9 2000.

[8] M. Scanlon, "Research guides: Micro enterprise: Data sources," 10 2008.

[9] B. H. Wixom, I. A. Someh, A. Zutavern, and C. M. Beath, "Explanation: a new enterprise data monetization capability for ai," 7 2020.

[10] N. Pushpalatha, S. Mohanram, S. Sivaranjani, and A. Prasanth, 5 Enterprise Data, pp. 95–114. De Gruyter, 5 2023.

[11] L. M. Lindgren, Storage Area Networks Meet Enterprise Data Networks, pp. 289–298. Auerbach Publications, 10 1999.

[12] C. Hanson, S. Nackerud, and K. L. Jensen, "Affinity strings: Enterprise data for resource recommendations," Code4Lib Journal, 12 2008.

[13] ENTERPRISE DATA STORAGE AND CORPORATE MEMORY FACILITY, pp. 217–238. CRC Press, 4 2016.

[14] S. Z. Ashraf, "Designing scalable data architectures for enterprise data platforms," International Journal For Multidisciplinary Research, vol. 5, 1 2023.

[15] A. Anant and R. Prasad, "Privacy preservation for enterprises data in edge devices," Journal of ICT Standardization, 2 2022.

[16] A. Leblanc, Enterprise Data Management with SAP NetWeaver MDM. 11 2007.

[17] R. D. Hackathorn, "Distributing enterprise data," Data Based Advisor archive, vol. 9, pp. 92–99, 11 1991.

[18] AMCIS - The Business Value of Enterprise Data Models, 7 2016.

[19] B. Williams, Enterprise Data Management. 1 2017.

[20] A. H. Murtaza, A Framework for Developing an Enterprise Data Warehousing Solution, pp. 755–764. Auerbach Publications, 12 2021.

[21] P. Ghavami, Chapter 2 Enterprise Data Governance Directive, pp. 24–36. De Gruyter, 11 2020.

[22] W. Keller, Enterprise Data Modeling Practices, pp. 161–170. Auerbach Publications, 9 2000.

[23] - Enterprise Data Anonymization Execution Model, pp. 104–109. Auerbach Publications, 5 2013.

[24] A. H. Murtaza, "A framework for developing enterprise data warehouses," Information Systems Management, vol. 15, pp. 21–26, 9 1998.